**OPERATING SYSTEMS LEC 06**

**Main Memory**

* Memory can be considered a large array of word/bytes each having an address
  + Repository of quickly accessible data shared by CPU and I/O devices
  + Memory/registers are only storage that CPU can directly access
* Main memory is a volatile storage device
* In a 32-bit system you can have access a maximum of 232 memory addresses
* Programs must be mapped to absolute addresses and loaded into memory
* OS is responsible for
  + Tracking which parts of memory are being used and by whom
  + Decide which processes to load next when memory space becomes available
  + Allocate and deallocate memory
* A program is bought onto memory to be executed
* Improving CPU utilisation and speed we store:
  + Share memory and hence need memory management scheme
  + Most allow user processes to reside in physical memory
    - Processes have separate memory areas to protect from other processes
* Addresses are generally symbolic such as variable *count*
  + Compiler will *bind* symbolic addresses to re-locatable addresses
  + Loader/linkage editor binds each relocatable address to an absolute address
    - Start is 10000, 14 counts later is 10014
* Binding is a mapping from one address space to another

**Binding Time**

* Compile time: if memory location is a known, absolute code can be generated
  + Must recompile if starting location changes
  + Process must be loaded at same place in memory
* Load time: binding is delayed until code is loaded into memory
  + Relocatable code is generated if memory location is unknown at compile
  + Process must reload if starting address changes
* Execution time: binding is delayed until run time
  + Processes can be moved during execution from one memory segment to another
  + Needs special hardware support for address mapping (base/limit registers)

**Logical vs. Physical Address Space**

* Logical address – generate by CPU (sometimes virtual addresses)
* Physical Address – address seen by the memory unit
* Logical/Physical addresses differ at in execution-time address binding scheme
  + Mapping is done by memory management unit (MMU) – hardware device

A screenshot of a cell phone

Description automatically generated**Memory-Management Unit (MMU)**

* In MMU, value of relocation register is added to addresses generated by user process when sent to memory.
  + Due to user processes dealing with logical addresses

**Dynamic Loading**

* Routing is not loaded until it is called
* For better memory-space utilisation
  + Unused routines are never loaded – kept in disk in relocatable format
* Useful with large amounts of code needing to handle infrequently occurring cases
* No special support from OS

**Dynamic Linking**

* Linking is delayed until execution time
* Used for system libraries
* Small piece of code, *stub*, is used to locate appropriate memory-resident library routine
* OS checks if routine is in process’ memory address

**Swapping**

* Processes can be swapped temporarily out of memory to *backing store* and brought back into memory for continued execution
  + Backing store is a large, fast disk storing copies of all memory images for all users with direct access to the memory images
* When CPU scheduler selects process to run, it calls dispatcher to check if process is in memory
  + If not, if there isn’t enough free memory, dispatch will swap out one or more processes from memory, freeing up space for process
* Major part of swap time is transfer time (File I/O)
* In compile/loading time binding a process will be swapped back into the same memory space that it used before
* Problem: Swapped out process is in middle of doing I/O
  + Fix by latching job in memory while involved in I/O
  + Do I/O only into OS buffers

**Contiguous Allocation**

* Main memory is divided into two partitions
  + Resident operating system, in low memory with interrupt vector
  + One or more user processes, held in high memory
* Contiguous allocation: each process is in one contiguous memory location

**Memory Mapping and Protection**

* Uses relocation and limit registers
  + Relocation register contains value of smallest physical address
  + Limit register contains range of logical addresses – each logical address must be less than the limit register
  + The relocation-register scheme is used to protect user processes from each other, and from accessing OS memory
  + The registers are loaded by dispatcher as part of context switch

**Multiple-Partition Allocation**

* Memory can be divided into fixed-sized partitions
* Memory can be divided into variable partitions
  + OS maintains information about allocated and free partitions(holes) in table
  + When process arrives, it is allocated memory from a hole large enough

**Dynamic Storage-Allocation Problem**

* How to satisfy request of size *n* from list of free holes
  + First fit: allocate first hole that is big enough
  + Best fit: Allocate smallest hole that is big enough
    - Must search entire list unless ordered
  + Worst fit: Allocate the largest hole
    - Also search entire list
* First fit and best fir are better than worst in terms of speed and storage utilisation

**Fragmentation**

* Storage-Allocation strategies suffer from external fragmentation
  + External fragmentation – total memory space exists to satisfy a request by is not contiguous
* Internal fragmentation – allocated memory may be slightly larger than requested
* Reduce external fragmentation by compaction:
  + Shuffle memory contents to place all free memory in a large block
  + Only possible if relocation is dynamic (done at execution time)
  + Must determine cost – size of memory contents to move (less the better)
  + Selection of strategy is difficult

**Paging**

* Produces no external fragmentation, may create internal fragmentation
  + Used by Unix, Linux, Windows
* In this system:
  + Logical address space of a process can be non-contiguous
  + Physical memory is divided into fixed-sized blocks called frames
  + Logical memory is divided into fixed-sized blocks called pages
    - Page size = frame size
  + Frame size is a power of 2; defined by hardware
    - Easier for page number and page offset simple
    - OS keeps track of free frames
* To run program of size *n* pages:
  + OS finds *n* free frames and loads program
  + Set up page table to translate logical to physical addresses

**Address translation scheme**

* Page number (p) – used as index for page table which contains base address for each page in physical memory
* Page offset(d) = combines with base address to define physical memory address that is sent to the memory unit
* For 2nd logical address space and page size of 2d, p=m-d
* 0010 would give frame zero, and then the 3rd (index 2) unit within the frame

**Implementation of page table**

* Each page table is kept in main memory
  + Page table register points to page table
  + Page table length register shows size of page table
* Every data/instruction access requires memory access for page table access and one for data/instruction access
  + Solved by using fast-lookup hardware cache called associative registers
    - Also called translation look-aside buffers (TLBs)
* TLB is expensive, so typical number of entries: 32-1024
  + One for data and one for instruction. The are also multi-level TLBs

**Associative Register**

* Address translation (A, A’)
  + Parallel search. VERY FAST
  + If A is in TLB get frame number
  + Otherwise get frame number from page table in memory
  + This step is done as part of instruction pipeline in CPU – adding no search time penalty.
* Some TBLs include Address Space Identifiers (ASIDs) in each TLB entry
  + An ASID identifies a process
  + ASID for running process must match the ASID in its TBL entry
  + Without this TBL must be flushed when there is a context switch

**Effective Access Time**

* **Notation:** 
  + Associative lookup = ε time unit.
  + Assume memory cycle time is *t* time unit.
  + Hit ratio = α is the percentage of times that a page number is found in the associative registers.
* Effective access time (EAT):
  + If the page number is in TLB: time = ε + *t.*
  + If the page number is NOT in TLB: time = ε + *t* + *t.*
  + EAT = (*t* + ε) α + (2 *t* + ε)(1 - α) .

**Memory Protection**

* Implemented by associating a protection bit with each frame
  + A valid-invalid bit is attached to each entry in page table
  + Valid shows associated page in process’ logical address space
  + Invalid shows page is not in process’ address space
* One bit is used to set page read-write or read-only access
* Each attempt to page is checked against protection bits
  + Trap CPU if illegal attempt

**Hierarchical Paging**

* Very large logical address systems create very big page tables
* One solution is to divide page table into smaller pieces
  + For two-level paging scheme, page itself is paged also

**Two-Level Paging**

* A logical address (32-bit machine, 4K page size) is divided into:
  + Page number consisting of 20 bits
  + Page offset consisting of 12 bits
* Since page table is paged, number is divided into:
  + 10-bit page number
  + 10-bit page offset
* Page number is index to outer page whilst offset is displacement within page of inner page table (from the outer page)
* Can take more memory accesses now!
* Catching permits performance to remain reasonable

A screenshot of a cell phone

Description automatically generated

**Hashed Page Table**

* Commonly used when address space > 32 bits
* Virtual page number hashed into page table
  + Chain of elements hashing to same location (in linked list)
* Virtual page number is compared in this chain searching for a match
* For address spaces > 64, use clustered page tables

**Inverted Page Table**

* In paging system, each process has a page table associated with it
  + Drawback: each table may consist of millions of entries using large amounts of physical memory – instead use inverted page table
* Only uses one table that has one entry for each real page of memory
* Every entry contains virtual address of page stored in the real memory location with information about the process that owns that page
* Scheme decreases memory needed to store each page table but increases time to search table when a page reference occurs
  + May use hash table to limit search to one – or a few page table entries

**Shared Pages**

* Advantage of paging is possibility of sharing common code (re-entrant code also called pure code)
* Re-entrant code is non-self-modifying code – never changes during execution
* Shared code:
  + One copy of read-only (re-entrant) code shared among processes
* Private code and data:
  + Each process keeps separate copy of code and data
  + Pages for private code and data can appear anywhere in logical address space

**Segmentation**

* Memory management with paging makes separation between user view and the actual physical memory.
* Segmentation is a memory-management scheme that supports user view of memory
* From user’s view, program is a collection of segments, which are logical unit such as:
  + Main program
  + Procedure/function
  + Local variables, global variables
  + Common block
  + Stack
  + Symbol table, arrays
* Segments have name and length; addresses specify both segment name and offset
* Program specifies each address by two quantities: segment name and offset (within segment)

**Segmentation Architecture**

* Logical address contains two tuple: <segment-number, offset>
* Segment table – maps 2-D user-defined addressed into one-dimensional physical addresses
  + Tables have:
    - Base – starting physical address where segments reside in memory
    - Limit – length of segment
* Segment-table base register points to segment table’s location in memory
* Segment-table length register indicated number of segments used by program
  + With trap if too big
* Sharing
  + Shared segments and segment number
* Allocation
  + Fist fit/ best fit
  + External fragmentation
* Protection
  + Validation bit and read/write/execute privileges
* Memory allocation is a dynamic storage-allocation problem